Linguagens e Ambientes de Programação (2011/2012)

Teórica 03 (01/Mar/2012)

Funções com múltiplos parâmetros. Formas curried e não-curried. Aplicação parcial de funções curried.

Associatividade do operador de aplicação. Associatividade do construtor de tipos funcionais "->". Formas equivalentes de escrita de funções.

Funções como valores de primeira classe. Exemplos.

Listas. Construtores de listas. Emparelhamento de padrões. Exemplos.

Funções com múltiplos argumentos em OCaml

Em OCaml há duas formas de escrever funções com mais do que um argumento: a **forma não-curried** e a **forma curried**. A segunda forma é mais elaborada e a que tem mais vantagem técnicas.

Forma não-curried

Os vários argumento agrupam-se num único tuplo ordenado. Exemplos:

```
let nAdd (x,y) = x+y ;;
nAdd: (int * int) -> int

let nAdd3 (x,y,z) = x+y+z ;;
nAdd3: (int * int * int) -> int

nAdd (3,4) = 7
nAdd3 (2,8,1) = 11
```

Tecnicamente, a função `nAdd` tem apenas um parâmetro, de tipo `int*int`. Mas claro, através desse único parâmetro conseguimos passar duas unidades de informação.

Forma curried

Os argumentos ficam separados. Exemplos:

```
let cAdd x y = x+y ;;
cAdd: int -> int -> int

let cAdd x y z = x+y+z ;;
cAdd: int -> int -> int -> int

cAdd 3 4 = 7
cAdd 2 8 1 = 11
```

Escrever *funções curried* não tem nada que saber: basta usar espaços em branco a separar os parâmetros, na cabeça de cada função. Contudo o tipo destas funções afigura-se surpreendente para quem o observa pela primeira vez. Isso é resultado da representação interna das funções em OCaml. Segue-se a explicação...

Representação interna das funções em OCaml

Por uma questão de simplicidade e regularidade de implementação, o OCaml só usa internamente funções anónimas com um único argumento. Uma função com múltiplos argumentos é convertida para um formato interno especial - chamado **forma interna** - que envolve apenas funções anónimas com um único argumento.

Por exemplo, a função

```
let cAdd x y = x+y ;;
```

é internamente convertida em:

```
let cAdd = fun x -> (fun y -> x+y) ;;
```

Repare na engenhosa a ideia que está por detrás do esquema de tradução.

Vejamos mais alguns exemplos de tradução, lado a lado:

```
let cAdd x y z = x+y+z ;;      let cAdd = fun x -> (fun y -> (fun
z -> x+y+z)) ;;

let f x = x+1 ;;               let f = fun x -> x+1 ;;

let nAdd (x,y) = x+y ;;        let nAdd = fun (x,y) -> x+y ;;
```

Avaliação de expressões envolvendo funções com múltiplos argumentos

Compare a avaliação das duas seguintes expressões:

```
nAdd (2,3)
= (fun (x,y) -> x+y) (2,3)
= 2 + 3
= 5

cAdd 2 3
= (fun x -> (fun y -> x+y)) 2 3
= (fun y -> 2+y) 3
= 2 + 3
= 5
```

Aplicação parcial

As funções curried têm a vantagem de poderem ser **aplicadas parcialmente** ou seja, poderem ser invocadas omitindo alguns dos argumentos do final. Eis um exemplo de aplicação parcial:

```
let succ = cAdd 1 ;;
succ: int -> int
```

Associatividades

- O **operador de aplicação** é associativo à esquerda. (Note que este operador é *invisível*, pois nunca se escreve).
  - Portanto, a expressão `f a b` deve ser interpretada como `(f a) b`.
- O **construtor de tipos funcionais ->** é associativo à direita.
  - Portanto, o tipo `int -> int -> int` deve ser interpretado como `int -> (int -> int)`.

Formas equivalentes de escrever a mesma função

As seguintes quatro declarações são equivalentes, no sentido em que declaram exatamente a **mesma** função `f:int->int->int`.

```
let f x y = x + y ;;                    (* formato externo
preferido *)
let f x = (fun y -> x+y) ;;             (* formato externo *)
let f = (fun x y -> x+y) ;;             (* formato externo *)
let f = (fun x -> (fun y -> x+y)) ;;    (* formato interno *)
```

Todas são convertidas para a mesma forma interna.

Funções como valores de primeira classe

Nas linguagens funcionais as funções têm o estatuto de **valores de primeira classe**. Isso significa que as funções têm um estatuto tão importante como o dos inteiros, reais, e outros tipos predefinidos.

Concretamente, numa linguagem funcional as funções podem ...

- Ser passadas como argumento para outras funções;
- Podem ser retornadas por outras funções;
- Podem ser usadas como elementos constituintes de estruturas de dados;
- Têm uma representação literal própria. Exemplo: `(fun x->x+1)`

Exemplos

Exemplo 1. Composição de funções.

```
let compose f g = fun x -> f (g x) ;;
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

Também se pode escrever:

```
let compose f g x = f (g x) ;;
```

Exemplo 2. Função para converter do formato não-curried para o formato curried.

```
let curry f = fun x -> fun y -> f (x,y) ;;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Também se pode escrever:

```
let curry f x y = f (x,y) ;;
```

Exercício: Escrever a função inversa `uncurry`.

Exercício: Que funções são as seguintes e quais os seus tipos:

- `uncurry compose`
- `compose curry uncurry`
- `compose curry uncurry`

Exemplo 3. Como representar conjuntos usando apenas funções?

Um conjunto é uma entidade cuja principal característica é a possibilidade de se poder saber se um valor lhe pertence ou não lhe pertence. Assim vamos representar cada conjunto por uma função booleana que aplicada a um valor produz `true` se esse valor pertence ao conjunto e `false` se não pertence. Esta é a chamada **função característica** do conjunto.

- Conjunto vazio:

```
let set0 = fun x -> false ;;
```

- Conjunto universal:

```
let setu = fun x -> true ;;
```

- Criação de conjunto singular:

```
let set1 x = fun y -> y = x ;;
```

Exercício: Usando esta representação, escrever as funções `belongs`, `union` e `intersection`.

Exemplo 4. Estrutura de dados com funções: lista de funções.

```
let mylist = [(fun x -> x+1); (fun x -> x*x)] ;;
```

Uma limitação das funções

No caso geral, saber se duas funções dadas são iguais (i.e. saber se produzem sempre os mesmos resultados para os mesmos argumentos) é um problema que não pode ser resolvido por computador. Numa tal situação, costuma dizer-se que o problema é indecidível.

```
# (fun x -> x+1) = (fun x -> x+1) ;;
Exception: Invalid_argument "equal: functional value".
```

Listas homogéneas em OCaml

Apresentam-se aqui os três aspetos essenciais relativos as listas em OCaml: como se escrevem listas literais; como se constroem novas listas a partir de listas mais simples; como se analisam listas e se extraem os seus elementos constituintes.

Listas literais

Exemplos de listas literais:

```
[]                                  : 'a list
[2;4;8;5;0;9]                       : int list
["ola"; "ole"]                       : string list
[[1;2]; [4;5]]                      : int list list
[(fun x -> x+1); (fun x -> x*x)]    : (int->int) list
```

Construtores de listas

Estão disponíveis dois construtores de listas que, como o nome indica, servem para construir listas novas:

```
[] : 'a list
:: : 'a -> 'a list -> 'a list
```

- O construtor `[]` chama-se "lista vazia" e representa a lista vazia.
- O construtor `::` chama-se "cons" e serve para construir listas não vazias.

O operador :: é associativo à direita.

Exemplos de utilização de `cons`:

```
2::[3;4;5] = [2;3;4;5]
1::2::3::[] = [1;2;3]
[]::[] = [[]]
[1;2]::[3;4] = ERRO
4::5 = ERRO
[1;2]::[[3;4;5]] = [[1;2];[3;4;5]]
```

Processamento de listas usando emparelhamento de padrões

O processamento de listas efetuar-se por *análise de casos*, usando a construção **match** e *padrões*. Exemplo:

```
(* len : 'a list -> int *)

let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

A função anterior trata dois casos, cada um dos quais tem um padrão diferente associado. Os vários casos são analisados sequencialmente, de cima para baixo.

Mais um exemplo. A seguinte função aplica-se a uma lista de pares ordenados e troca entre si as componentes de cada par:

```
(* swapPairs : ('a * 'b) list -> ('b * 'a) list *)

let rec swapPairs l =
    match l with
      [] -> []
    | (x,y)::xs -> (y,x)::swapPairs xs
;;
```

Na próxima aula estudaremos um método sistemático que nos ajudará a escrever funções recursivas sobre listas.

#140